Re-implementing a Ruby on Rails Site in Yesod

CS 557: Functional Languages @ Portland State University
Dylan Laufenberg, winter 2019

1 Introduction

For the last seven years or so, I've been developing and expanding a Ruby on Rails-powered website for a Sacramento, CA soup kitchen named Sharing God's Bounty. I've had mixed feelings about my choice of Ruby on Rails over the last half-dozen years, so this final project seemed to be an excellent opportunity to experiment with a significantly different Web technology stack than any other that I've ever used. I've implemented a narrow, vertical slice of functionality to experience and experiment with Haskell across many aspects of server-side code for the Web. I use the Yesod web framework for my project, and I provide comparisons to my Ruby on Rails experience as appropriate through this report. I followed the quick start instructions here, which provided me with Yesod 1.6.0.3.

Yesod's documentation comes primarily from the Yesod book [1] (available online here), so a large portion of my work for this project has simply been reading through the book to understand how to utilize this highly sophisticated, mature framework. I don't attempt to introduce or define every concept in Yesod here. Instead, I highlight particularly noteworthy applications of Haskell to the domain of Web application development and discuss how I use Yesod to build out the features I develop for this final project. In particular, I assume the reader knows about Haskell features such as Template Haskell, QuasiQuotes, and language pragmas either through prior knowledge or by reading the Haskell section of the Yesod book [2].

New section: Cover "The Basics"?

2 Resources and type-safe URIs

To represent **routes**, which are mappings from URIs to server-side code, Yesod uses what it calls resources. A **resource** is a `data` constructor that acts as the canonical name of the route in views [1]. For instance, the default site scaffold includes a static route of the form:

```
/static StaticR Static appStatic
```

This line specifies a `/static` route, a resource constructor named `StaticR` as the name of the route, and a subsite named `Static` whose function `appStatic` will handle requests on this route [3]. In order to create a link to a static file, e.g. an image served as part of a template, we invoke the `StaticR` constructor with an appropriate argument. The `Static` subsite generates "static file identifiers" at compile time [4]. When we insert a link using a resource constructor:

```
<img ... src="@{StaticR img_logo_png}" ... />
```

Yesod generates a correct link to the static resource at compile-time. In this case, the user's browser receives:

```
<img ... src="http://localhost:3000/static/img/logo.png?etag=GVVUzjtL" ... />
```

2.1 Benefits of Resources

There are two primary benefits to this approach, both of which leverage Haskell features from class (albeit at a very advanced level).

Check links at compile time Yesod uses resources to generate correct links at compile-time, which allows it to check whether said links are valid and raise compile-time errors if any problems are detected. In this case, the Static subisite checks the identifier `img_logo_png` against all generated static file identifiers. If it finds a match, it inserts the correct link. If not, then GHC raises an error. For instance, if we mistakenly specify a JPG logo:

```
<img ... src="@{StaticR img_logo_jpg}" ... />
```

GHC will detect the issue for us and (rather obtusely) notify us of the issue with a compile-time error:

```
    • Variable not in scope: img_logo_jpg :: Route Static
    • Perhaps you meant ‘img_logo_png’ (imported from Import.NoFoundation)
    |
163 |             $(widgetFile "default-layout")
    |               ^^^^^^^^^^^^^^^^^^^^^^^^^^
```

In effect, Yesod automatically tests every internal link at compile time, without the need to manually write tests against the generated HTML. We, the developers, need to test our functions for generating links, then Yesod takes care of the rest for us. In this case, I feel comfortable as a developer taking it on faith that the static file subsite is well-tested.

Refactor routes with confidence Generating links at compile time also eliminates an entire class of common and nefarious errors: broken links. Updating or moving routes is a common occurrence in Web application development, even with careful planning. Yesod makes this safe and automatic by generating all such links from resource constructors at compile time. To change the route across the entire application, all we need to do is change the route definition:

```
/dynamic StaticR Static appStatic
```

And all of our generated links to that route will automatically update when we recompile:

```
<img ... src="http://localhost:3000/dynamic/img/logo.png?etag=GVVUzjtL" ... />
```

3 Shakespearean Templates

Yesod provides domain-specific languages (DSLs) over HTML, CSS, and Javascript, all named after Shakespearean characters. In converting my site, I migrated the HTML, CSS, and Javascript, in that order. (For the full, reference versions of all information in this section that's not otherwise cited, see [5].)

All of Yesod's DSLs provide a common set of interpolation features:

- **Variable interpolation** All variables in scope when a template is rendered are available through variable interpolation. For instance, in my main site template, I write

  ```
  <title>#{pageTitle pc} - Sharing God's Bounty
  ```

  to insert the value of `pageTitle pc` as part of the HTML title.

- **Resource interpolation** We can insert type-safe URIs via `@{routeR}` resource interpolation, as discussed in Resources and type-safe URIs.

- **Template interpolation** We can embed templates in other templates by writing `^{templateName}`. This is safe in that it only allows embedding of templates of the same type and, of course, provides all the usual compile-time guarantees for both templates. **Add reference to Mixins section?**

3.1 Hamlet

Yesod uses Hamlet as its HTML DSL. Hamlet is similar to HTML, with a few distinguishing features.

Significant whitespace Hamlet templates infer closing tags from indentation levels. When you write, for instance, `<div>`, everything that should appear inside this tag must be nested below it. A matching closing tag is automatically inserted at the correct place in the generated HTML.

Class and ID shorthand Hamlet templates provide shorthand for specifying classes and IDs by prefacing a class or ID name with a `.` or a `#`, respectively. This shorthand even works with multiple classes, by writing either `<div .foo .bar>` or `<div ."foo bar">`. Either form generates the final HTML `<div class="foo bar"></div>`. To be honest, I tried these forms with the intention of exposing a shortcoming of Hamlet, and I was shocked to see that it handled both cases beautifully!

Example To see these features in action, let's take a look at my mobile navigation menu. The Hamlet representation is as follows: **UPDATE THIS EXAMPLE with the "final" version! It'll be WAY more impressive!**

```
<nav .mobile>
  <ul>
    <li>Menu items go here.
  <h2>
    Menu
```

Which compiles to:

```
<nav class="mobile"><ul><li>Menu items go here. </li>
</ul>
<h2>Menu</h2>
</nav>
```

Shortcomings There are, unfortunately, a number of shortcomings to Hamlet. First and perhaps most importantly, none of the four templating DSLs provide the strong, compile-time guarantees of correctness that we expect in Haskell, even though Hamlet in particular feels like it could. As a new user, I was genuinely surprised to learn that Hamlet doesn't raise compile-time errors for invalid tags! For instance, when I mistakenly wrote `<image ...>`, Hamlet happily generated both opening and closing `image` tags, which, understandably, confused both Firefox and me.

Second, although Hamlet knows that some tags like `img` are self-closing, it doesn't insert the closing `/` at the ends of such tags.

Third, Hamlet *does not* check whether classes and IDs written in Hamlet notation exist in linked Cassius or Lucius files!

Fourth, the HTML produced seems to be both arbitrary and virtually unreadable in its structure. The navigation example above is Hamlet's output, verbatim, and this is with minimization disabled! What particularly irks me about this oversight is that Hamlet *outright requires* proper indentation and spacing! Why it doesn't simply carry this indentation through to the final HTML is beyond my comprehension.

3.2 Lucius

Topics:

- Hamlet: went pretty smoothly, Initially had to cut out lots of features as NYI.
- Lucius/Cassius: oh, man. Whole subsection(s), whole big thing.

4 Reflections after the basics

Reflect on Yesod from the perspective of someone who's just made it through the basics. In particular, TinyOS's observation about being hard for beginners. I feel like I have to be an expert to do almost anything. Every piece of code feels like it's written for experts. And the lack of an accessible API reference (with good documentation) so far seems to be frustrating me.

References

[1] Michael Snoyman. *Developing Web Applications with Haskell and Yesod*. 1st edition. O'Reilly, 2012. URL: https://www.yesodweb.com/book.

[2] Michael Snoyman. "Developing Web Applications with Haskell and Yesod". In: 1st edition. O'Reilly, 2012. Chap. Haskell. URL: https://www.yesodweb.com/book/haskell.

[3] Michael Snoyman. "Developing Web Applications with Haskell and Yesod". In: 1st edition. O'Reilly, 2012. Chap. Routing and Handlers. URL: https://www.yesodweb.com/book/routing-and-handlers.

[4] Michael Snoyman. "Developing Web Applications with Haskell and Yesod". In: 1st edition. O'Reilly, 2012. Chap. Scaffolding and the Site Template. URL: https://www.yesodweb.com/book/scaffolding-and-the-site-template.

[5] Michael Snoyman. "Developing Web Applications with Haskell and Yesod". In: 1st edition. O'Reilly, 2012. Chap. Shakespearean Templates. URL: https://www.yesodweb.com/book/shakespearean-templates.